Runtime randomization and perturbation for virtual machines.

JAVIER CABRERA ARTEAGA

Licentiate Thesis in [Research Subject - as it is in your ISP]
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden [2022]

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

TRITA-ICT XXXX:XX
ISBN XXX-XX-XXXX-XXX-X

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av licentiatexamen i [ämne/subject] [veckodag/weekday] den [dag/day] [månad/month] [år/2022] klockan [tid/time] i [sal/hall], Electrum, Kungl Tekniska högskolan, Kistagången 16, Kista.

Tryck: Universitetsservice US AB

Abstract

Write your abstract here...
Keywords: Keyword1, keyword2, ...

Sammanfattning

Write your Swedish summary (popular description) here...

Keywords: Keyword1, keyword2, ...

Acknowledgements

Write your professional acknowledgements here...

Acknowledgements are used to thank all persons who have helped in carrying out the research and to the research organizations/institutions and/or companies for funding the research.

Name Surname,
Place, Date

Contents

Chapter 1

Introduction

Write a short introduction here...

1.1 Motivation

1.1.1 Why variants ?

1.1.2 Research questions

1. RQ1. To what extent can we artifically generate program variants for WebAssembly?
2. RQ2. To what extent are the generated variants dynamically different?
3. RQ3. To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?

1.2 Contributions

Chapter 2

Background & State of the art

This chapter discusses state of the art in the areas of *WebAssembly* and *Software Diversification*. In Section 2.1 we discuss the WebAssembly language, its motivation, how WebAssembly binaries are generated, language specification and security related issues. In Section 2.2, we present a summary of Software Diversification and its foundational concepts. In addition, we compare and locate our contributions againts state of the art related works. We select the discussed works by their novelty, critical insights and representativeness of their used techniques. We finalize the chapter by stating our novel contributions.

2.1 WebAssembly overview

Over the past decades, JavaScript has been used in the majority of the browser clients to allow client-side scripting. However, due to the complexity of this language and to gain in performance, several approaches appeared, supporting different languages in the browser. For example, Java applets were introduced on web pages late in the 90's, Microsoft made an attempt with ActiveX in 1996 and Adobe added ActionScript later on 1998. All these attempts failed to persist, mainly due to security issues and the lack of consensus on the community of browser vendors.

In 2014, Emscripten proposed with a strict subset of JavaScript, asm.js, to allow low level code such as C to be compiled to JavaScript itself. Asm.js was first implemented as an LLVM backend. This approach came with the benefits of having all the ahead-of-time optimizations from LLVM, gaining in performance on browser clients [39] compared to standard JavaScript code. The main reason why asm.js is faster, is that it limits the language features to those that can be optimized in the LLVM pipeline or those that can be directly translated from the source code. Besides, it removes the majority of the dynamic characteristics of the language, limiting it to numerical types, top-level functions, and one large array in the memory directly accessed as raw data. Since asm.js is a subset of JavaScript it was compatible with all engines at that moment. Asm.js demonstrated that client-code could be improved with the right language design and standarization. The work of Van Es et al. [30] proposed to shrink JavaScript to asm.js in a source-to-source strategy, closing the cycle and extending the fact that asm.js was mainly a compilation target for C/C++ code. Despite encouraging results, JavaScript faces several limitations related to the characteristics of the language. For example, any JavaScript engine requires the parsing and the recompilation of the JavaScript code which implies significant overhead.

Following the asm.js initiative, the W3C publicly announced the WebAssembly (Wasm) language in 2015. WebAssembly is a binary instruction format for a stack-based virtual machine and was officially stated later by the work of Haas et al. [29] in 2017. The announcement of WebAssembly marked the first step of standarizing bytecode in the web environment. Wasm is designed to be fast, portable, self-contained and secure, and it outperforms asm.js [29]. Since 2017, the adoption of WebAssembly keeps growing. For example; Adobe, announced a full online version of Photoshop[1] written in WebAssembly; game companies moved their development from JavaScript to Wasm like is the case of a fully Minecraft version [2]; and the case of Blazor [3], a .Net virtual machine implemented in Wasm, able to execute C# code in the browser.

From source to Wasm

All WebAssembly programs are compiled ahead of time from source languages. LLVM includes Wasm as a target since version 8, supporting a broad range of frontend languages such as C/C++, Rust, Go or AssemblyScript[4]. The resulting binary, works similarly to a traditional shared library, it includes instruction codes, symbols and exported functions. In Figure 2.1, we illustrate the workflow from the creation of Wasm binaries to their execution in the browser. The process starts by compiling the source code program to Wasm (Step ①). This step includes ahead-of-time optimizations. For example, if the Wasm binary is generated out of the LLVM pipeline, all optimizations in the LLVM

The step ② includes the standard library to Wasm as JavaScript code. This code includes the external functions that the Wasm binary needs for its execution inside the host engine. For example, the functions to interact with the DOM of the HTML page are imported in the Wasm binary during its call from the JavaScript code. The standard library can be manually written, however, compilers like Emscripten, Rust and Binaryen can generate it automatically, making this process completely transparent to developers.

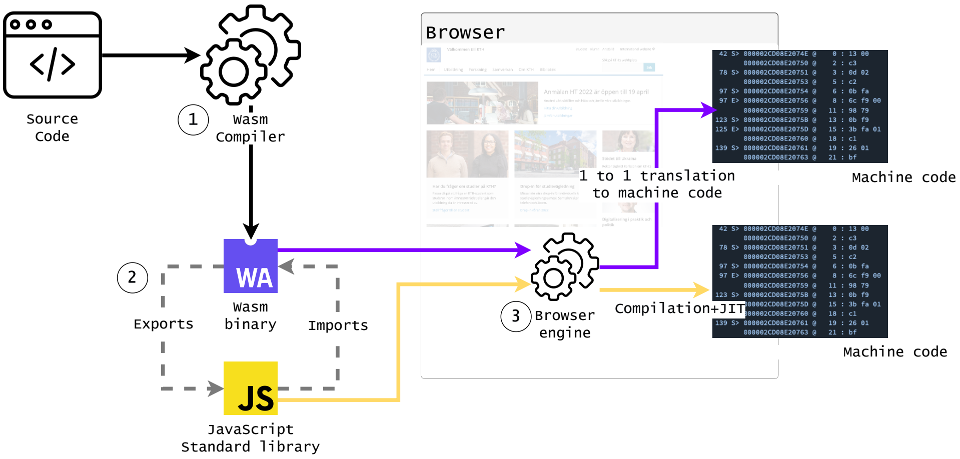
[1] https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw

[2] https://satoshinm.github.io/NetCraft/

[3] https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor

[4] subset of the TypeScript language

Finally, the third step (Step ③), includes the compilation and execution of the client-side code. Most of the browser engines compile either the Wasm and JavaScript codes to machine code. In the case of JavaScript, this process involves JIT and hot code replacement during runtime. For Wasm, since it is closer to machine code and it is already optimized, this process is a one-to-one mapping. For instance, in the case of V8, the compilation process only applies simple and fast optimizations such as constant folding and dead code removal. Once V8 completes the compilation process, the generated machine code for Wasm is final and is the same used along all its executions. This analysis was validated by conversations with the V8's dev team and by experimental studies in our previous contributions.



```
 42 S> 000002CD08E2074E @    0 : 13 00
       000002CD08E20750 @    2 : c3
 78 S> 000002CD08E20751 @    3 : 0d 02
       000002CD08E20753 @    5 : c2
 97 S> 000002CD08E20754 @    6 : 0b fa
 97 E> 000002CD08E20756 @    8 : 6c f9 00
       000002CD08E20759 @   11 : 98 79
123 S> 000002CD08E2075B @   13 : 0b f9
125 E> 000002CD08E2075D @   15 : 3b fa 01
       000002CD08E20760 @   18 : c1
139 S> 000002CD08E20761 @   19 : 26 01
       000002CD08E20763 @   21 : bf
```

```
 42 S> 000002CD08E2074E @    0 : 13 00
       000002CD08E20750 @    2 : c3
 78 S> 000002CD08E20751 @    3 : 0d 02
       000002CD08E20753 @    5 : c2
 97 S> 000002CD08E20754 @    6 : 0b fa
 97 E> 000002CD08E20756 @    8 : 6c f9 00
       000002CD08E20759 @   11 : 98 79
123 S> 000002CD08E2075B @   13 : 0b f9
       000002CD08E2075D @   15 : 3b fa 01
       000002CD08E20760 @   18 : c1
139 S> 000002CD08E20761 @   19 : 26 01
       000002CD08E20763 @   21 : bf
```

Figure 2.1: WebAssembly building, compilation in the host engine and execution.

Wasm can execute directly and is platform independent, making it useful for IoT and Edge computing [3, 15]. For instance, Cloudflare and Fastly adapted their platforms to provide Function as a Service (FaaS) directly with WebAssembly. In this case the standard library, instead of JavaScript, is provided by any other language stack that the host environment supports. In 2019, the bytecode alliance team [5] proposed WebAssembly System Interface (WASI). WASI is the foundation to build Wasm code outside of the browser with a POSIX system interface platform. It standarizes the adoption of WebAssembly outside web browsers [17] in heterogeneous platforms.

WebAssembly specificities

WebAssembly defines its own Instruction Set Architecture (ISA) [26]. It is an abstraction close to machine code instructions but agnostic to CPU architectures. Thus, Wasm is platform independent. The ISA of Wasm includes also the necessary

[5] https://bytecodealliance.org/

components that the binary requires to run in any host engine. A Wasm binary has a unique module as its main component. A module is composed by sections, corresponding to 13 types, each of them with an explicit semantic and a specific order inside the module. This makes the compilation to machine code faster.

In Listing 2.1 and Listing 2.2 we illustrate a C program and its compilation to Wasm. The C function contains: heap allocation, external function declaration and the definition of a function with a loop, conditional branching, function calls and memory accesses. The code in Listing 2.2 is in the textual format for the generated Wasm. The module in this case first defines the signature of the functions(Line 2, Line 3 and Line 4) that help in the validation of the binary defining its parameter and result types. The information exchange between the host and the Wasm binary might be in two ways, exporting and importing functions, memory and globals to and from the host engine (Line 5, Line 35 and Line 36). The definition of the function (Line 6) and its body follows the last import declaration at Line 5.

The function body is composed by local variable declarations and typed instructions that are evaluated in a virtual stack (Line 7 to Line 32 in Listing 2.2). Each instruction reads its operands from the stack and pushes back the result. The result of a function call is the top value of the stack at the end of the execution. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add` at Line 32. A valid Wasm binary should have a valid stack structure that is verified during its translation to machine code. The stack validation is carried out using the static types of Wasm. As the listing shows, instructions are annotated with a type. Wasm binaries contain only four datatypes, `i32`, `i64`, `f32` and `f64`.

Wasm manages the memory in a restricted way.A Wasm module has a linear memory component that is accessed as `i32` pointers and should be isolated from the virtual stack. The declaration of the linear data in the memory is showed in Line 37. The memory access is illustrated in Line 15. This memory is usually bound in browser engines to 2Gb of size, and it is only shareable between the process that instantiate the Wasm binary and the binary itself (explicitly declared in Line 33 and Line 36). Therefore, this ensures the isolation of the execution of Wasm code.

Wasm also provides global variables in their four primitive types. Global variables (Line 34) are only accessible by their declaration index, and it is not possible to dynamically address them. For functions, Wasm follows the same mechanism, either the functions are called by their index (Line 30) or using a static table of function declarations. This latter allows modeling dynamic calls of functions (through pointers) from languages such as C/C++, however, the compiler should populate the static table of functions.

Wasm control flow structures are different from standard assembly programs. All instructions are grouped into blocks, being the start of a function the root block. Two consecutive block declarations can be appreciated in Line 10 and Line 11 of Listing 2.2. Control flow structures jump between block boundaries and not to any position in the code like regular assembly code. A block may specify the state that the stack must have before its execution and the result stack

Listing 2.1: Example C function.

```
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi(12.54);
    }

    return A[0];
}
```

Listing 2.2: WebAssembly code for Listing 2.1.

```
1  (module
2    (type (;0;) (func (param f32) (result i32)))
3    (type (;1;) (func))
4    (type (;2;) (func (result i32)))
5    (import "env" "ftoi" (func $ftoi (type 0)))
6    (func $main (type 2) (result i32)
7      (local i32 i32)
8      i32.const -1000
9      local.set 0 ;loop iteration counter;
10     block  ;label = @1;
11       loop  ;label = @2;
12         i32.const 0
13         local.get 0
14         i32.add
15         i32.load
16         local.tee 1
17         i32.const 101
18         i32.ge_s ;loop iteration condition;
19         br_if 1 ;@1;
20         local.get 0
21         i32.const 4
22         i32.add
23         local.tee 0
24         br_if 0 ;@2;
25       end
26       i32.const 0
27       return
28     end
29     f32.const 0x1.9147aep+3 ;=12.54;
30     call $ftoi
31     local.get 1
32     i32.add)
33   (memory (;0;) 1)
34   (global (;4;) i32 (i32.const 1000))
35   (export "memory" (memory 0))
36   (export "A" (global 2))
37   (data $data (0) "\00\00\00\00...")
38 )
```

value coming from its instructions. Inside the Wasm binary the blocks explicitly define where they start and end (Line 25 and Line 28). By design, each block executes independently and cannot execute or refer to outer block values. This is quarantieed by explicitly annotating the state of the stack before and after the block. Three instructions handle the navigation between blocks: unconditional break, conditional break (Line 19 and Line 24) and table break. Each breaking block instruction can only jump the execution to one of its enclosing blocks. For example, in Listing 2.2, Line 19 forces the execution to jump to the end of the first block at Line 10 if the value at the top of the stack is greater than zero.

We want to remark that de description of Wasm in this section follows the version 1.0 of the language and not its proposals for extended features. We follow those features implemented in the majority of the vendors according to the Wasm roadmap [27]. On the other hand we excluded instructions for datatype conversion, table accesses and the majority of the arithmetic instructions for the sake of simplicity.

WebAssembly's security

As we described, WebAssembly is deterministic and well-typed, follows a structured control flow and explicitly separates its linear memory model, global variables and the execution stack. This design is robust [16] and makes easy for compilers and engines to sandbox the execution of Wasm binaries. Following the specification of Wasm for typing, memory, virtual stack and function calling, host environments should provide protection against data corruption, code injection, and return-oriented programming (ROP).

However, WebAssembly is vulnerable under certain conditions, at the execution engine's level [21]. Implementations in both browsers and standalone runtimes [3] are vulnerable. Genkin et al. demonstrated that Wasm could be used to exfiltrate data using cache timing-side channels [24]. One of our previous contributions trigger a CVE on the code generation component of wasmtime, highlighting that even when the language specification is meant to be secure, its execution might not be. Moreover, binaries itself can be vulnerable. The work of Lehmann et al. [14] proved that C/C++ source code vulnerabilities can propagate to Wasm such as overwriting constant data or manipulating the heap using stackoverflow. Even though these vulnerabilities need a specific standard library implementation to be exploited, they make a call for better defenses for WebAssembly. Several proposals for extending WebAssembly in the current roadmap could address some existing vulnerabilities. For example, having multiple memories could incorporate more than one memory, stack and global spaces, shrinking the attack surface. However, the implementation, adoption and settlement of the proposals are far from being a reality in all browser vendors.

2.2 Software Diversification

The low presence of defenses implementations for WebAssembly motivates our work on Software Diversification as a preemptive technique that can help against known and yet unknown vulnerabilities. Software Diversification has been widely studied in the past decades, this section discusses its state of the art. Software diversification consists in synthesizing, reusing, distributing and executing different, functionally equivalent programs. We use the concept of functional equivalence defined in the seminal work of Cohen et al. [63] as input output equivalence. Two programs are equivalent if, given identical input, they produce the identical output.

According to the survey of Baudry and Monperrus [38], the motivation for software diversification can be separated in five categories: reusability [56], software testing [49], performance [46], fault tolerance [64] and security [63]. Our work contributes to the latter two categories:

(G1) *Fault tolerance and reliability:* Mainly refers to the implementation of independent yet functionally equivalent programs for the sake of consensus during execution. Different programs are deployed and executed at the same time, the final result is selected from all computation results. If some programs fail, the system is still able to respond. During decades, this same idea was applied to hardware, for example in airplanes is common to have more than one sensor providing the same function.

(G2) *Security:* Mainly refers to break code reuse attacks [62] by using diverse functional programs. The main idea is to change the observable behavior of a program by changing its version every time is invoked. Thus, attackers cannot get the same information from a different source. For example, Crane et al. [37] hardened power side-channels by using diversification of software. On the same topic, Roy et al. [18] use preexisting machine learning algorithms to defeat adversarial-like attackers.

Software diversification sources.

There are two main sources of software diversification: natural(ND) and artificial diversity(AD) [38]. Natural diversity can be controlled or can be an unpredicted consequence of developing processes. Controlled natural diversity is usually called Design Diversity or N-Version Diversity. It is addressed using engineering decisions [64]. In practice, it consists of providing N development teams with the same requirements. The teams develop N independent versions, using different approaches. On the other hand, Natural Diversity can emerge from the spontaneous software development processes. To illustrate the Natural Diversity phenomenon, CodeForces[6] shows more than 350 different and successful solutions in C++ for a single requirements based problem in a single programming contest. Software market is an expected source of natural diversity. Sengupta et al. [28] used this fact to reach the security goal (G2).

Notice that Natural Diversity can rely on itself to escalate, and it is coped by the preexistence of software. This can be seen as a limitation. For example, in the context o this work, the natural diversity for WebAssembly programs is nearly inexistence [6]. When natural diversity is not enough, it is innate to think that the source for diversification needs to be artificial. Automated software diversification consists on artificially synthesize software.

According to the seminal work of Cohen et al. [63] automatic software diversification can be reached by mutation strategies. A mutation strategy is a set of rules to define how a specific piece of software should be changed to provide a different, yet functionally equivalent, variant. A mutation can be appplied at different layers of software lifecycle, from compilation to execution and from source code to executable binary. We have found that the foundation for automatic software diversity has barely changed since Cohen in 1993. Complemented with

[6] `https://codeforces.com/contest/1667/status/page/2?order=BY_PROGRAM_LENGTH_ASC`

the work of Baudry and Monperrus [38], we enumerate the strategies for automatic software diversification.

(S1) *Equivalent arithmetic instructions* Numeric calculations can be expressed theoretically in an infinite number of ways. This strategy is simple but powerful since the complexity of program variants dramatically increases. In terms of overhead, the size of the program variant increases with the size of the replacement.

(S2) *Instruction reordering* This strategy reorder instructions or entire program blocks if they are independent to each other. This strategy generates program variants without affecting its size and execution time.

(S3) *Variable substitution* This strategy changes the location of variable declarations. It has lower impact on low level programs unless compilers resort symbol tables. It prevents static examination an analysis of parameters and alter memory locations. The strategy should not affect the size of program variants neither their execution time.

(S4) *Adding, changing, removing jumps* This strategy creates program variants by adding, changing or removing jumps inside the original program. At high level, this can be reached by loop splitting or by inserting arbitrary branching. This strategy increases the execution time of variants.

(S5) *Adding, changing, removing calls* This strategy is similar to the previous one (S4). It extends the same idea by adding function calls inside the stack. It is mainly implemented by inlining and de-inlining expressions inside the program. When an expression is inlined, then its original instruction is replaced by a function call that perform the same calculation of the original subexpression.

(S6) *Garbage insertion* This strategy adds instructions to the program that are independent of the original sequence of instructions. It extends S1 by supporting more instructions like random memory accesses. Dealing with code-reuse attacks, Homescu et al. [42] propose inserting garbage NOP instructions directly in LLVM IR to generate a variant with a different code layout at each compilation. Jackson et al. [45] have explored how to use NOP operations inserted during compiling time to statically diversify programs. Jackson et al. [45] used the optimization flags of several compilers to generate semantically equivalent binaries out of the same source code. These techniques place the compiler at the core of the diversification technique.

(S7) *Program layout randomization in memory and stack* This strategy is usually implemented over compilers. The generation of the final binary and how it operates its memory is randomized.

(S8) *ISA randomization* This strategy encodes the original program, for example by using a simple XOR operation over its binary bytestream. This technique is strong against attacks involving the examination of code. It should not affect the size of program variants or their execution times. The encoding/decoding operations are performed only once. Seminal works include instruction-set randomization [58, 60] to create a unique mapping between artificial CPU instructions and real ones.

(S9) *Simulation* It is an interpretation mechanism similar to encoding (S9) but the execution of programs is delegated to a custom interpreter instead of using preexisting execution hosts. The program is decoded at runtime every time it is invoked. Chew, and Song [61] target operating system randomization. They randomize the interface between the operating system and the user applications: the system call numbers, the library entry points (memory addresses), and the stack placement. All those techniques are dynamic, done at runtime using load-time preprocessing and rewriting.

(S10) *Intermixing* With the existence of more than one program variant their execution can be mixed. The decision of which variant executes is decided at runtime. This strategy is the core for randomization, multivariant execution and the execution by consensus defined in G1. This strategy is complex to implement because the integrity of the memory and stack needs to be stable between programs.

Usages of Software Diversity

We categorize the applications of software diversity in three main usages.

(U1) *N-Version:* More than one independent program variant executed in one single machine. For example, the work of Sengupta et al. [28] proposed to change indistinctly between database engines and backend server implementations in a monolitic web application. Their approach decreases the reach of exploitable CVEs from request to request. In this area, Coppens et al. [43] use compiler transformations to iteratively diversify software. Their work aims to prevent reverse engineering of security patches for attackers targeting vulnerable programs. Their approach continuously applies a random selection of predefined transformations using a binary diffing tool as feedback [?] . A downside of their method is that attackers are, in theory, able to identify the type of transformations applied and find a way to ignore or reverse them. Bathkar et al. [59, 57] proposed three kinds of randomization transformations: randomizing the base addresses of applications and libraries memory regions, random permutation of the order of variables and routines, and the random introduction of random gaps between objects. Dynamic randomization can address different kinds of problems. In particular, it mitigates an extensive range of memory error exploits. Recent work in this field includes stack layout randomization against data-oriented programming [20], and memory safety violations [4], as well as a technique to reduce the exposure time of persistent memory objects to increase the frequency of address randomization [11].

(U2) *Randomization:* One program that decides at runtime which of its variants to execute. Couroussé et al. [32] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. A randomization technique creates a set of unique executions for the very same program [59]. Previous works have attempted to generate diversified variants that are alternated during execution. It has been shown to drastically increase the number of execution traces required by a side-channel attack as Amarilli et al. [48] has shown. On the same topic, Agosta et al. [40] and Crane et al. [37] modify the LLVM toolchain to compile multiple functionally equivalent variants to randomize the control flow of software at runtime.

(U3) *Multivariant Execution(MVE):* Multiple program variants composed in one single binary that can execute either separately depending on external configuration or in parallel for consensus computation. In 2006, security researchers at University of Virginia laid the foundations of a novel approach to security that consists in executing multiple variants of the same program, [55]. Bruschi et al. [54] and Salamat et al. [53] pioneered the idea of executing the variants in parallel. Subsequent techniques focus on Multivariant Execution for mitigating memory vulnerabilities [22] and other specific security problems incl. return-oriented programming attacks [33] and code injection [47]. A key design decision of MVE is whether it is achieved in kernel space [19], in user-space [50], with exploiting hardware features [31], or even through code polymorphism [25]. Finally, one can neatly exploit the limit case of executing only two variants [44, 35]. Notably, Davi et al. proposed Isomeron [36], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. With this strategy a potential attacker cannot predict whether the original or the variant of a program will execute. Researching on MVE in a distributed setting like the Edge [?] has been less researched. Voulimeneas et al. proposed a multivariant execution system by parallelizing the execution of the variants in different machines [2] for the sake of efficiency.

U1, U2 and U3 can be categorized as Moving Target Defense strategies. Moving Target Defense for software was first proposed as a collection of techniques that aim to improve the security of a system by constantly moving its vulnerable components [10, 41]. Usually, MTD techniques revolve around changing system inputs and configurations to reduce attack surfaces. This increases uncertainty for attackers and makes their attacks more difficult. Ultimately, potential attackers cannot hit what they cannot see. MTD can be implemented in different ways, including via dynamic runtime platforms [18].

Statement of Novelty

We contribute to Software Diversification for WebAssembly using Artificial Diversification, for N-Version, Randomization and Multivariant Execution usages

(U1, U2, U3) for the sake of reliability and security (G1 and G2). The main motivation for our contributions is that we see in WebAssembly a monoculture problem, if one environment is vulnerable, all the others are vulnerable in the same manner as the same WebAssembly binary is replicated. Besides, the WebAssembly environment lacks natural diversity [38]. Compared to the work of Harrand et al. [?] , in WebAssembly, one could not use preexisting and different program versions to provide diversification. The current limitations on security and the lack of preexisting diversity motivate our work on software diversification as one preemptive mitigation among a wide range of security countermeasures.

In Table 2.1 we listed related work on Software Diversification that support our work. The table is composed by the authors and the reference to their work, the source of diversification (natural or artificial), followed by one column for each motivation, strategy and usage (G1, G2, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, U1, U2 and U3). Each cell in the table contains a checkmark if the strategy, the motivation or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order. The last two rows locate our contributions.

TODO The table is incomplete

As the table illustrates, we push the state of the art on Software Diversification. Our first contribution, CROW [9] generates multiple program variants for WebAssembly using the LLVM pipeline. It contributes to the state of the art in artificially creating diversity for WebAssembly. While the number of related work for software diversity is large, only one approach has been applied to the context of WebAssembly. To the best of our knowledge, the closest diversification work on the browsers involving WebAssembly is the work of Romano et al. [1]. They proposed to de-inline JavaScript (S5) subexpressions and replace them with function calls to WebAssembly counterparts. The presence of two different engines, one for JS and another for WebAssembly in the majority of the browser vendors motivated their work. They empirically demonstrated that malware classifiers could be evaded with this diversification technique. On the other hand, WebAssembly is a novel technology and the adoption of defenses in for it are still under development [3, 5].

CROW, extrapolates the idea of superdiversification [52] for WebAssembly. CROW works directly with LLVM IR, enabling it to generalize to more languages and CPU architectures, something not possible with the x86-specific approach of previous works. CROW focuses on the static diversification of software. However, because of the specificities of code execution in the browser, this is not far from being a randomization approach. For example, since WebAssembly is served at each page refreshment, every time a user asks for a WebAssembly binary, she can be served a different variant provided by CROW. It also can be used in fuzzing campaigns [?] to provide reliability. The diversification created by CROW can unleash hidden behaviors in compilers and interpreters. Thanks to CROW, a bug was discovered in the Lucet compiler [7]

[7] https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem

Authors	**ND**	**AD**	**G1**	**G2**	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**S10**	**U1**	**U2**	**U3**
Chew and Song [61]		✓							✓							✓	
Kc et al. [58]		✓		✓								✓	✓				
Barrantes et al. [60]		✓		✓								✓				✓	
Bhatkar et al. [59]		✓		✓		✓	✓			✓	✓					✓	
Bhatkar et al. [57]		✓		✓		✓	✓			✓	✓					✓	
Cox et al. [55]	✓			✓										✓			✓
Bruschi et al. [54]		✓		✓							✓			✓		✓	
Salamat et al. [53]		✓		✓					✓					✓			✓
Jacob et al. [52]		✓			✓		✓			✓					✓		
Salamat et al. [50]																	
Amarilli et al. [48]				✓	✓								✓			✓	
Jackson [45]		✓								✓							
Maurer et al. [44]																✓	
Homescu et al. [42]		✓								✓							
Coppens et al. [43]	✓																
Agosta et al. [40]	✓			✓													✓
Crane et al. [37]		✓		✓		✓			✓	✓				✓			✓
Davi et al. [36]		✓		✓										✓		✓	
Couroussé et al. [32]				✓		✓				✓			✓	✓		✓	
Koning et al. [31]																✓	
Sengupta et al. [28]	✓			✓										✓			✓
Lu et al. [22]		✓		✓							✓			✓			✓
Belleville et al. [25]																✓	
Aga et al. [20]		✓		✓							✓					✓	
Roi et al. [18]	✓			✓										✓			✓
Österlund et al. [19]																	
Xu et al. [11]		✓		✓							✓					✓	
Lee et al. [4]		✓		✓							✓					✓	
Voulimeneas et al. [2]		✓												✓			✓
Cabrera Arteaga et al. [9]		✓		✓	✓	✓	✓	✓		✓					✓	-	
Cabrera Arteaga et al. [8]		✓		✓	✓	✓	✓	✓	✓	✓				✓		✓	✓

Table 2.1: Comparison table of related work.

Moreover, with MEWE [8], we contribute to the field of randomization at two stages. First, we automatically generate variants of a given program with CROW [9]. Second, we randomly select which variant is executed at runtime, creating a multivariant execution scheme that randomizes the observable behaviors at each run of the program. We use the natural redundancy of Edge-Cloud computing architectures to deploy an internet-based MVE.

Conclusions

In this chapter we presented the background on the WebAssembly language including our motivation on its security issues and related work enforcing its security. Software Diversification has been widely researched, not being the case for WebAssembly. In this chapter, we aim to settle down the foundation to study automatic diversification for WebAssembly. We stated our contributions to the field of artificial diversity by extending current state of the art. Our contributions are obtained by following the methodology described in Chapter 3.

Chapter 3

Methodology

In this chapter, we present our methodology to answer the research questions enunciated in Subsection 1.1.2. We investigate three research questions. In the first question, **TODO** we artificially generate WebAssembly program **TODO** This is the how, the why is more important variants and quantitatively compare the static differences between variants. Our second research question focuses on comparing their behavior during their execution. The final research question evaluates the feasibility of using the program variants in security-sensitive environments. We evaluate our generated program variants in an Edge-Cloud computing platform proposing a novel multivariant execution approach.

The main objective of this thesis is to study the feasibility of automatically creating program variants out of preexisting program sources. To achieve this objective, we use **TODO** too generic: the empirical method [13] , proposing a solution and evaluating it through quantitative analyzes in case studies. We follow an iterative and incremental approach on the selection of programs for our corpora. To build our corpora, we find a representative and diverse set of programs to generalize, even when it is unrealistic following an empirical approach, as much as possible our results. We first enunciate the corpora we share along this work to answer our research questions. Then, we establish the metrics for each research question, set the configuration for the experiments, and describe the protocol.

Corpora

Our experiments assess the impact of artificially created diversity. The first step is to build a suitable corpus of programs' seeds to generate the variants. Then, we answer all our research questions with three corpora of diverse and representative programs for our experiments. **TODO** elaborate on diverse and representative We build **TODO** why three: tell the reader here it feels like a magic number our three corpora in an escalating strategy. The first corpus is diverse and contains simple programs in terms of code size, making them easy to manually analyze.

The second corpus is a project meant for security-sensitive applications. The third corpus is a QR encoding decoding algorithm. The work of Hilbig et al. [6] shows that approximately 65% of all WebAssembly programs come out of C/C++ source code through the LLVM pipeline, and more than 75% if the Rust language is included. Therefore, all modules in the corpora are considered to come along the LLVM pipeline. In the following, we describe the filtering and description of each corpus.

1. **Rosetta** : We take programs from the Rosetta Code project[1]. This website hosts a curated set of solutions for specific programming tasks in various programming languages. It contains many tasks, from simple ones, such as adding two numbers, to complex algorithms like a compiler lexer. We first collect all C programs from the Rosetta Code, representing 989 programs as of 01/26/2020. We then apply several filters: the programs should successfully compile and, they should not require user inputs to automatically execute them, the programs should terminate and should not result in non-deterministic results.

   The result of the filtering is a corpus of 303 C programs. All programs include a single function in terms of source code. These programs range from 7 to 150 lines of code and solve a variety of problems, from the *Babbage* problem to *Convex Hull* calculation.

2. **Libsodium**: This project is encryption, decryption, signature, and password hashing library implemented in 102 separated modules. The modules have between 8 and 2703 lines of code per function. This project is selected based on two main criteria: first, its importance for security-related applications, and second, its suitability to collect the modules in LLVM intermediate representation.

3. **QrCode**: This project is a QrCode and MicroQrCode generator written in Rust. This project contains 2 modules having between 4 and 725 lines of code per function. As Libsodium, we select this project due to its suitability for collecting the modules in their LLVM representation. Besides, this project increases the complexity of the previously selected projects due to its integration with the generation of images.

In Table 3.1 we listed the corpus name, the number of modules, the total number of functions, the range of lines of code, and the original location of the corpus. TODO Add commit sha1 for libsodium and qrcode.

[1] http://www.rosettacode.org/wiki/Rosetta_Code

Corpus	No. modules	No. functions	LOC range	Location
Rosetta	-	303	7 - 150	`https://github.com/KTH/slumps/tree/master/benchmark_programs/rossetta/valid/no_input`
Libsodium	102	869	8 - 2703	`https://github.com/jedisct1/libsodium`
QrCode	2	1849	4 - 725	`https://github.com/kennytm/qrcode-rust`
Total		3021		

Table 3.1: Corpora description. The table is composed by the name of the corpus, the number of modules, the number of functions, the lines of code range and the location of the corpus.

3.1 RQ1. To what extent can we artifically generate program variants for WebAssembly?

TODO the main issue with "Methoology for RQ1" is that CROW has not been introcuded / cast as a contribution yet

This research question investigates whether we can artificially generate program variants for WebAssembly. We use CROW to generate variants from an original program, written in C/C++ in the case of Rosetta corpus and LLVM bitcode modules in the case of Libsodium and QrCode. In Figure 3.1 we illustrate the workflow to generate WebAssembly program variants. We pass each function of the corpora to CROW as a program to diversify. To answer RQ1, we study the outcome of this pipeline, the generated WebAssembly variants.

Metrics

To assess our approach's ability to generate WebAssembly binaries that are statically different, we compute the number of variants and the number of unique variants for each original function of each corpus. On top, we define the aggregation of these former two values to quantitatively evaluate RQ1 at the corpus level.

We start by defining what a program's population is. This definition can be applied in general to any collection of variants of the same program. All definitions are based upon bytecodes and not the source code of the programs.

Definition 1. *Program's population $M(P)$:* Given a program P and its generated variants v_i, the program's population is defined as.

$$M(P) = \{v_i \text{ where } v_i \text{ is a variant of P}\}$$

Notice that, the program's population includes the original program P.

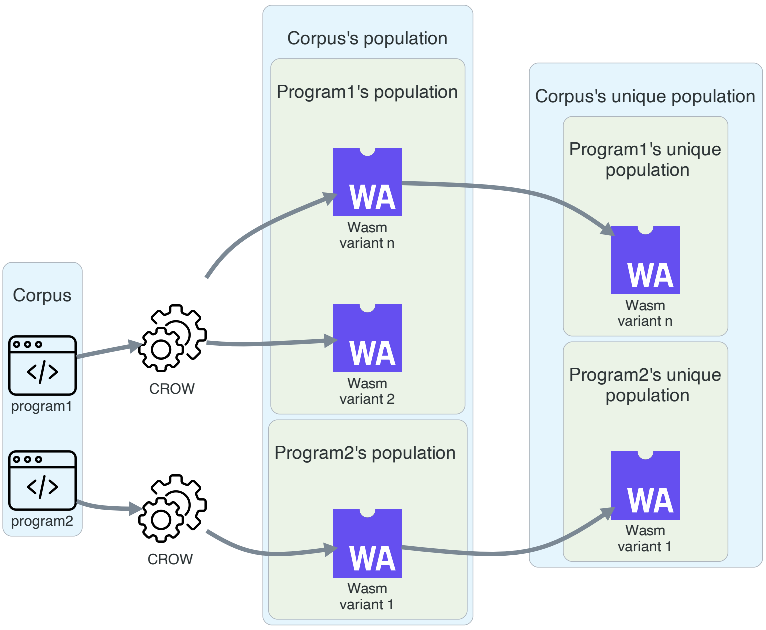


Figure 3.1: The program variants generation for RQ1.

Beyond the program's population, we also want to compare how many program variants are unique. The subset of unique programs in the program's population hints how the variants are different between them and not only against the original program. For example, imagine a program P with two program variants V_1 and V_2, the program population is composed by $\{P, V_1$ and $V_2\}$ where V_1 is different from P, and V_2 is different from P. Either, if V_1 is equal or different from V_2, the program's population still be the same.

Definition 2. *Program's unique population $U(P)$:* Given a program P and its program's population $M(P)$, the program's unique population is defined as.

$$U(P) = \{v \ \in \ M(P)\}$$

such that $\forall v_i, v_j \in U(P)$, $md5sum(v_i) \neq md5sum(v_j)$. $Md5sum(v)$ is the md5 hash calculated over the bytecode stream of the program file v. Notice that, the original program P is included in $U(P)$.

Metric 1. *Program's population size $S(P)$:* Given a program P and its program's population $M(P)$ according to Definition 1, the program's population size is defined as.

$$S(P) = |M(P)|$$

Metric 2. *Program's unique population size $US(P)$:* Given a program P and its program's unique population $U(P)$ according to Definition 2, the program's unique population size is defined as.

$$US(P) = |U(P)|$$

Metric 3. *Corpus population size $CS(C)$:* Given a program's corpus C, the corpus population size is defined as the sum of all program's population sizes over the corpus C.

$$CS(C) = \Sigma S(P) \ \forall \ P \ \in \ C$$

Metric 4. *Corpus unique population size $UCS(C)$:* Given a program's corpus C, the corpus unique population size is defined as the sum of all program's unique population sizes over the corpus C

$$UCS(C) = \Sigma US(P) \ \forall \ P \ \in \ C$$

Protocol

TODO text sounding like "background text": To generate program variants, we synthesize program variants with an enumerative strategy, checking each synthesis for equivalence modulo input [23] against the original program. An enumerative synthesis is a brute-force approach to generate program variants. With a maximum number of instructions, it constructs and checks all possible programs up to that limit. For a simplified instance, with a maximum code size of 2 instructions in a programming language with L possible constructions, an enumerative synthesizer builds all $L \times L$ combinations finding program variants. For obvious reasons, this space is nearly impossible to explore in a reasonable time as soon as the limit of instructions increases. Therefore, we use two parameters to control the size of the search space and hence the time required to traverse it. On the one hand, one can limit the size of the variants. On the other hand, one can limit the set of instructions used for the synthesis. In our experiments for RQ1, we use all the 60 supported instructions in our synthesizer.

The former parameter allows us to find a trade-off between the number of variants that are synthesized and the time taken to produce them. For the current evaluation, given the size of the corpus and the properties of its programs, we set the exploration time to 1 hour maximum per function for Rosetta . In the cases of Libsodium and QrCode, we set the timeout to 5 minutes per function. The decision behind the usage of lower timeout for Libsodium and QrCode is motivated by the properties listed in Table 3.1. The latter two corpora are remarkably larger regarding the number of instructions and functions count.

We pass each of the 303 + 869 + 1849 functions in the corpora to CROW, as Figure 3.1 illustrates, to synthesize program variants. We calculate the *Corpus population size*(Metric 3) and *Corpus unique population size*(Metric 4) for each corpus and conclude by answering RQ1.

3.2 RQ2. To what extent are the generated variants dynamically different?

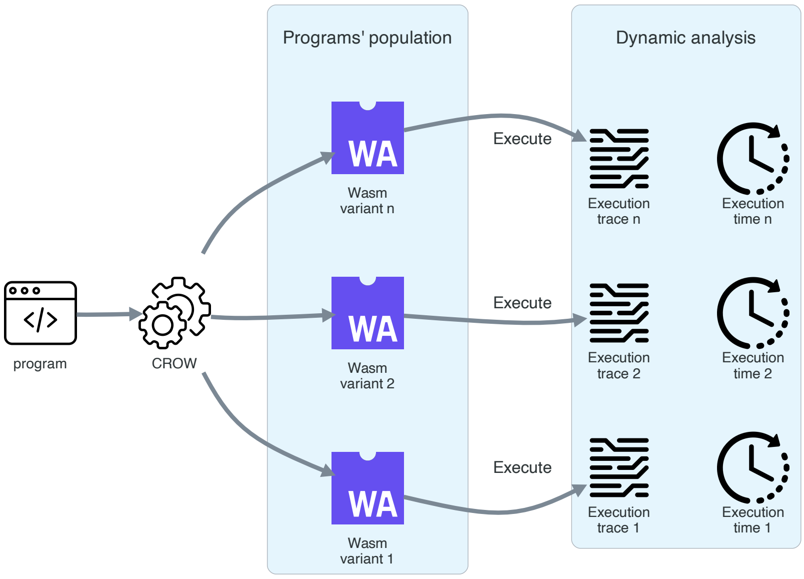


Figure 3.2: Dynamic analysis for RQ2.

In this second research question, we investigate to what extent the artificially created variants are dynamically different between them and the original program. To conduct this research question, we could separate our experiments into two fields as Figure 3.2 illustrates: static analysis and dynamic analysis. The static analysis focuses on the appreciated differences among the program variants, as well as between the variants and the original program, and we address it in answering RQ1.

With RQ2, we focus on the last category, the dynamic analysis of the generated variants. This decision is supported because dynamic analysis complements RQ1 and, it is essential to provide a full understanding of diversification. We use the original functions from Rosetta corpus described in Section 3 and their variants generated to answer RQ1. We use only Rosetta to answer RQ2 because this corpus is composed of simple programs that can be executed directly without user interaction, *i.e.,* we only need to call the interpreter passing the WebAssembly binary to it.

TODO Motivate, Increasing attack surface does not necesarilly has an impact on defence. For example, reordering instructions for code blocks that are never executed does not impact attacker

To dynamically compare programs and their variants, we execute each program on each programs' population to collect and execution times. We define execution trace and execution time in the following section.

Metrics

We compare the execution traces of two any programs of the same population with a global alignment metric. We propose a global alignment approach using Dynamic Time Warping (DTW). Dynamic Time Warping [65] computes the global alignment between two sequences. It returns a value capturing the cost of this alignment, which is a distance metric. The larger the DTW distance, the more different the two sequences are. DTW has been used for comparing traces in different domains. For software, De A. Maia et al. [51] proposed to identify similarity between programs from execution traces. In our experiments, we define the traces as the sequence of the stack operations during runtime, *i.e.,* the consecutive list of `push` and `pop` operations performed by the WebAssembly engine during the execution of the program. In the following, we define the $TraceDiff$ metric.

TODO before, define this and give an illutrative listing plus, says how you collect those traces, that's part of the protocol: between the stack operation traces

Metric 5. *TraceDiff:* Given two programs P and P' from the same program's population, TraceDiff(P,P'), computes the DTW distance collected during their execution.
A TraceDiff of 0 means that both traces are identical. The higher the value, the more different the traces.

Moreover, we use the execution time distribution of the programs in the population to complement the answer to RQ2. For each program pair in the programs' population, we compare their execution time distributions. We define the execution time as follows:

Metric 6. *Execution time:* Given a WebAssembly program P, the execution time is the time spent to execute the binary.

Protocol

To compare program and variants behavior during runtime, we analyze all the unique program variants generated to answer RQ1 in a pairwise comparison using the value of aligning their execution traces (Metric 5). We use SWAM[2] to execute each program and variant to collect the stack operation traces. SWAM is a WebAssembly interpreter that provides functionalities to capture the dynamic information of WebAssembly program executions, including the virtual stack operations.

Furthermore, we collect the execution time, Metric 6, for all programs and their variants. We compare the collected execution time distributions between programs using a Mann-Withney U test [67] in a pairwise strategy.

3.3 RQ3. To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?

TODO The last method is too short

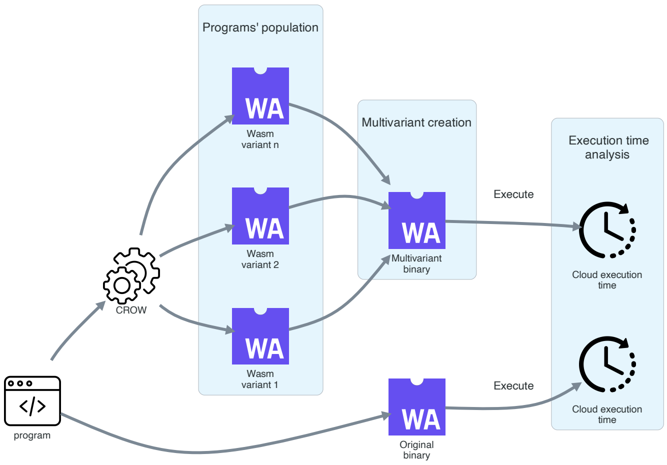


Figure 3.3: Multivariant binary creation and workflow for RQ3 answering.

In the last research question, we study whether the created variants can be used in real-world applications and what properties offer the composition of the variants as multivariant binaries. We build multivariant binaries (according to Definition 3), and we deploy and execute them at the Edge. The process of *mixing*

[2] `https://github.com/satabin/swam`

multiple variants into one multivariant binary is an essential contribution of the thesis that is presented in details in [7]. RQ3 focuses on analyzing the impact of this contribution on execution times and timing side-channels. To answer RQ3, we use the variants generated for the programs of Libsodium and QrCode corpora, we take $2 + 5$ programs interconnecting the LLVM bitcode modules (mentioned in Table 3.1). We illustrate the protocol to answer RQ3 in Figure 3.3 starting from the creation of the programs' population.

Metrics

To answer RQ3, we build multivariant WebAssembly binaries meant to provide path execution diversification. In the following we define what a multivariant binary is.

Definition 3. *Multivariant binary:* Given a program P with functions $\{F_1, F_2, ..., F_n\}$ for which we generate function's populations (according to Definition 1) $\{M(F_1), M(F_2), ..., M(F_n)\}$. A multivariant binary B is the composition of the functions populations following the function call graph of P where each call $F_i \rightarrow F_j$ is replaced by a call $f \rightarrow g$, where $f \in M(F_i)$ and $g \in M(F_j)$ are function variants randomly selected at runtime.

We use the execution time of the multivariant binaries to answer RQ3. We use the same metric defined in Metric 6 for the execution time of multivariant binaries.

Protocol

We answer RQ3 by analyzing a real-world scenario. Since Wasm binaries are currently adopted for Edge computing, we run our experiments for RQ3 on the Edge. Edge applications are designed to be deployed as isolated HTTP services, having one single responsibility that is executed at every HTTP request. This development model is known as serverless computing, or function-as-a-service [12, 3]. We deploy and execute the multivariant binaries as end-to-end HTTP services on the Edge and we collect their execution times. To remove the natural jitter in the network, the execution times are measured at the backend space, *i.e.,* we collect the execution times inside the Edge node and not from the client computer. Therefore, we instrument the binaries to return the execution time as an HTTP header.

We do this process twice, for the original program and its multivariant binary. We deploy and execute the original and the multivariant binaries on 64 edge nodes located around the world. In Figure 3.4 we illustrate the word wide location of the edges nodes.

We collect 100k execution times for each binary, both the original and multivariant binaries. The number of execution time samples is motivated by the seminal work of Morgan et al. [34]. We perform a Mann-Withney U test [67] to compare both execution time distributions. If the P-value is lower than 0.05, the two compared distributions are different.

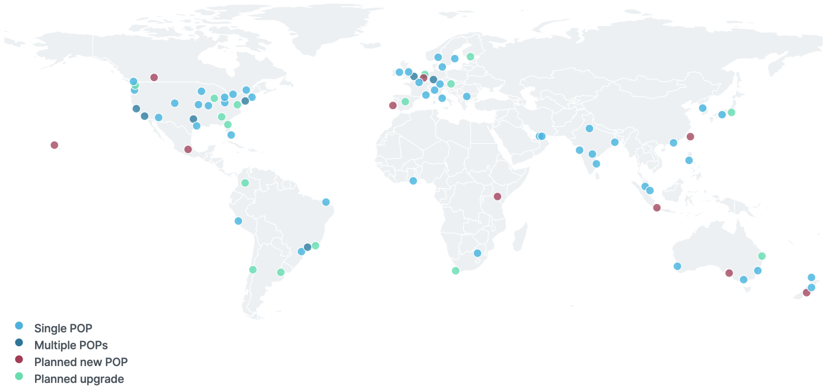


Figure 3.4: Screenshot taken from the Fastly Inc. platform used in our experiments for RQ3. Blue and darker blue dots represent the edge nodes used in our experiments.

Conclusions

This chapter presents the methodology we follow to answer our three research questions. We first describe and propose the corpora of programs used in this work. We propose to measure the ability of our approach to generate variants out of 3021 functions of our corpora. Then, we suggest using the generated variants to study to what extent they offer different observable behavior through dynamic analysis. We propose a protocol to study the impact of the composition variants in a multivariant binary deployed at the Edge. Besides, we enumerate and enunciate the properties and metrics that might lead us to answer the impact of automatic diversification for WebAssembly programs. In the next chapter, we present and discuss the results obtained with this methodology.

Chapter 4

Results

In this chapter, we sum up the results of the research of this thesis. We illustrate the key insights and challenges faced in answering each research question. To obtain our results, we followed the methodology formulated in Chapter 3.

4.1 RQ1. To what extent can we artifically generate program variants for WebAssembly?

As we describe in Section 3.1, our first research question aims to answer how to artifically generate WebAssembly program variants. This section is organized as follows. First we present the general results calculating the *Corpus population size*(Metric 3) and *Corpus unique population size*(Metric 4) for each corpus. Second, we discuss the challenges and limitations in program variants generation. Finally, we illustrate the most common code transformations performed by our approach and answer RQ1.

Program's populations

We summarize the results in Table 4.1. The table illustrates the corpus name, the number of functions to diversify, the number of successfully diversified functions (functions with at least one artificially created variant), the cumulative number of variants (*Corpus population size*) and the cumulative number of unique variants (*Corpus unique population size*).

We produce at least one unique program variant for 239/303 single function programs for Rosetta with one hour for a diversification timeout. For the rest of the programs (64/303), the timeout is reached before CROW can find any valid variant. In the case of Libsodium and QrCode, we produce variants for 85/869 and 32/1849 functions respectively, with 5 minutes per function as timeout. The rest of the functions resulted in timeout before finding function variants or produce no variants. For all programs in all corpora, we achieve 356/3021 successfully diversified functions, representing a 11.78% of the total. As the four and fifth columns show, the number of artifically created variants and the number of unique variants are larger than the original number of functions by one order of magnitude. In the case of Rosetta , the corpus population size is close to one million of programs.

TODO Elaborate on uniqueness...

TODO LOW: add histogram on variant sizes

Corpus	#Functions	# Diversified	# Variants	# Unique Variants
Rosetta	303	239	809900	2678
Libsodium	869	85	4272	3805
QrCode	1849	32	6369	3314
	3021	356	820541	9797

Table 4.1: General program's populations statistics. The table is composed by the name of the corpus, the number of functions, the number of succesfully diversified functions, the cumulative number of generated variants and the cumulative number of unique variants.

Challenges for automatic diversification

We have observed a remarkable difference between the number of successfully diversified functions versus the number of failed-to-diversify functions (third column of Table 4.1). Our approach successfully diversified 239/303, 85/869 and 32/1849 of the original functions for Rosetta , Libsodium and QrCode respectively. The main reason of this phenomenon is the set timeout for CROW.

We have noticed a remarkable difference between the number of diversified functions for each corpus, 809900 programs for Rosetta 4272 for Libsodium and 6369 for QrCode. The corpus population size for Rosetta is two orders of magnitude larger compared to the other two corpora. The reason behind the large number of variants for Rosetta is that, after certain time, our approach starts to combine the code replacements to generate new variants. However, looking at the fifth column, the number of unique variants have the same order of magnitude for all corpora. The variants generated out of the combination of several code replacements are not necessarily unique. Some code replacements can dominate over others, generating the same WebAssembly programs.

A low timeout offers more unique variants compared to the population size despite the low number of diversified functions, like the Libsodium and QrCode cases. This happens because, CROW first generates variants out of single code replacements and then starts to combine them. Thus, more unique variants are generated in the very first moments of the diversification process with CROW.

Apart from the timeout and the combination of variants phenomena, we manually analyze programs, searching for properties attempting to the generation of program variants using CROW. We identify another challenge for diversification. We have observed that our approach searches for a constant replacement for more than 45% of the instructions of each function while constant values cannot be inferred. For instance, TODO should be first briefly defined for the reader: constant values cannot be inferred. can you elaborate on the importance of constant values for diversification. for memory load operations because our tool is oblivious to a memory model.

Properties for large diversification

We manually analyzed the programs to study the critical properties of programs producing a high number of variants. This reveals one key factor that favors many unique variants: the presence of bounded loops. In these cases, we synthesize variants for the loops by replacing them with a constant, if the constant inferring is successful. Every time a loop constant is inferred, the loop body is replaced by a single instruction. This creates a new, statically different program. The number of variants grows exponentially if the function contains nested loops for which we can successfully infer constants.

A second key factor for synthesizing many variants relates to the presence of arithmetic. The synthesis engine used by our approach, effectively replaces arithmetic instructions with equivalent instructions that lead to the same result. For example, we generate unique variants by replacing multiplications with additions or shift left instructions (Listing 4.1). Also, logical comparisons are replaced, inverting the operation and the operands (Listing 4.2). Besides, our implementation can use overflow and underflow of integers to produce variants (Listing 4.3), using the intrinsics of the underlying computation model.

Listing 4.1: Diversification through arithmetic expression replacement.

```
local.get 0
i32.const 2
i32.mul
```

```
local.get 0
i32.const 1
i32.shl
```

Listing 4.2: Diversification through inversion of comparison operations.

```
local.get 0
i32.const 10
i32.gt_s
```

```
i32.const 11
local.get 0
i32.le_s
```

Listing 4.3: Diversification through overflow of integer operands.

```
i32.const 2
i32.mul
```

```
i32.const 2
i32.mul
i32.const
  -2147483647
i32.mul
```

TODO Add an example and refer to Cohen when machine code is generated, show that jumps are changed

Answer to RQ1.

We can provide diversification for 11.78% of the programs in our corpora. Constant inferring, complemented with the high presence of arithmetic operations and bounded loops in the original program increased the number of program variants.

4.2 RQ2. To what extent are the generated variants dynamically different?

Our second research question investigates the differences between program variants at runtime. To answer RQ2, we execute each program/variant generated to answer RQ1 for Rosetta corpus to collect their execution traces and execution times. For each programs' population we compare the stack operation traces (Metric 5) and the execution time distributions (Metric 6) for each program/variant pair.

This section is organized as follows. First, we analyze the programs' populations by comparing the traces for each pair of program/variant with TraceDiff of Metric 5. The pairwise comparison will hint at the results at the population level. We analyze not only the differences of a variant regarding its original program, we also compare the variants against other variants. Second, we do the same pairwise strategy for the execution time distributions Metric 6, performing a Mann-Withney U test for each pair of program/variant times distribution. Finally, we conclude and answer RQ2.

Stack operation traces.

In Figure 4.1 we plot the distribution of all comparisons (in logarithmic scale) of all pairs of program/variant in each programs' population. All compared programs are statically different. Each vertical group of blue dots represents all the pairwise comparison of the traces (Metric 5) for a program of Rosetta corpus for which we generate variants. Each dot represents a comparison between two programs' traces according to Metric 5. The programs are sorted by their number of variants in descending order. For the sake of illustration, we filter out those programs for which we generate only 2 unique variants.

We have observed that in the majority of the cases, the mean of the comparison values is remarkably large. We analyze the length of the traces, and one reason behind such large values of TraceDiff is that some variants result from constant inferring. For example, if a loop is replaced by a constant, instead of several symbols in the stack operation trace, we observe one. Consequently, the distance between two program traces is significant.

In some cases, we have observed variants that are statically different for which TraceDiff value is zero, *i.e.,* they result in the same stack operation trace. We

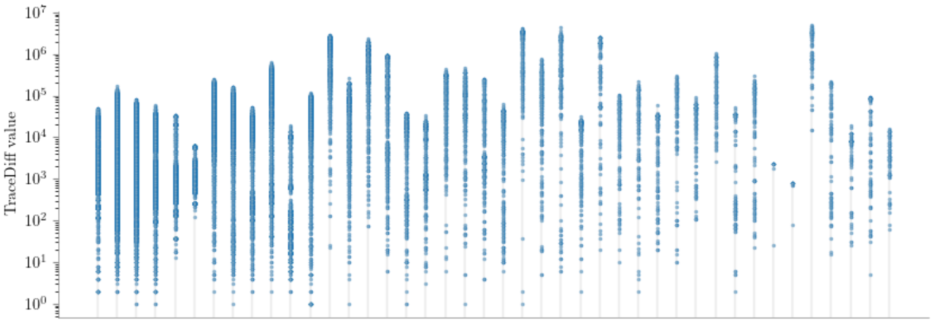


Figure 4.1: Pairwise comparison of programs' population traces in logarithmic scale. Each vertical group of blue dots represents a programs' population. Each dot represents a comparison between two program execution traces according to Metric 5.

identified two main reasons behind this phenomenon. First, the code transformation that generates the variant targets a non-executed or dead code. Second, some variants have two different instructions that trigger the same stack operations. For example, the code replacements below illustrate the case.

```
(1) i32.lt_u        i32.lt_s        (3) i32.ne            i32.lt_u
(2) i32.le_s        i32.lt_u        (4) local.get 6       local.get 4
```

In the four cases, the operators are different (original in gray color and the replacement in green color) leaving the same values for equal operands. The (1) and (2) cases are comparison operations leaving the value 0 or 1 in the stack taking into account the sign of their operands. In the third case, the replacement is less restricted to the original operator, but in both cases, the codes leave the same value in the stack. In the last case, both operands load a value of a local variable in the stack, the index of the local variable is different but the value of the variable that is appended to the trace is the same in both cases.

Execution times.

TODO recall again that execution time diversification is important, should the reader have forgotten about this.

Even when two programs of the same population offer different execution traces, their execution times can be similar (statistically speaking). In practice, the execution traces of WebAssembly programs are not necessarily accessible, being not the case with the execution time. For example, in our current experimentation we need to use our own instrumentation of the execution engine to collect the stack trace operations while the execution time is naturally accessible in any execution environment. This mentioned reasoning enforces our comparison of the execution times for the generated variants. For each program's population, we compare the execution time distributions, Metric 6, of each pair of program/variant. Overall diversified programs, 169 out of 239 (71%) have at least one variant with a different execution time distribution than the original program (P-value < 0.01 in the Mann-Withney test). This result shows that we effectively generate variants that yield significantly different execution times.

By analyzing the data, we observe the following trends. First, if our tool infers control-flows as constants in the original program, the variants execute faster than the original, sometimes by one order of magnitude. On the other hand, if the code is augmented with more instructions, the variants tend to run slower than the original.

In both cases, we generate a variant with a different execution time than the original. Both cases are good from a randomization perspective since this minimizes the certainty a malicious user can have about the program's behavior. Therefore, a deeper analysis of how this phenomenon can be used to enforce security will be discussed in answering RQ3.

To better illustrate the differences between executions times in the variants, we dissect the execution time distributions for one programs' population of Rosetta . The plots in Figure 4.2 show the execution time distributions for the `Hilbert curve` program and their variants. We illustrate time diversification with this program because, we generate unique variants with all types of transformations previously discussed in Section 4.1. In the plots along the X-axis, each vertical set of points represents the distribution of 100000 execution times per program/variant. The Y-axis represents the execution time value in milliseconds. The original program is highlighted in green color: the distribution of 10000 execution times is given on the left-most part of the plot, and its median execution time is represented as a horizontal dashed line. The median execution time is represented as a blue dot for each execution time distribution, and the vertical gray lines represent the entire distribution. The bolder gray line represents the 75% interquartile. The program variants are sorted concerning the median execution time in descending order.

For the illustrated program, many diversified variants are optimizations (blue dots below the green bar). The plot is graphically clear, and the last third represents faster variants resulting from code transformations that optimize the original program. Our tool provides program variants in the whole spectrum of time executions, lower and faster variants than the original program. The developer is in charge of deciding between taking all variants or only the ones providing the same or less execution time for the sake of performance. Nevertheless, this result calls for using this timing spectrum phenomenon to provide binaries with unpredictable execution times by combining variants. The feasibility of this idea will be discussed in Section 4.3.

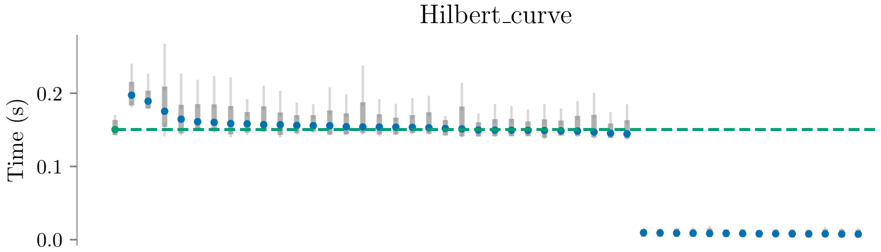


Figure 4.2: Execution time distributions for `Hilber_curve` program and its variants. Baseline execution time mean is highlighted with the magenta horizontal line.

Answer to RQ2.

We empirically demonstrate that our approach generates program variants for which execution traces are different. We stress the importance of complementing static and dynamic studies of programs variants. For example, if two programs are statically different, that does not necessarily mean different runtime behavior. There is at least one generated variant for all executed programs that provides a different execution trace. We generate variants that exhibit a significant diversity of execution times. For example, for 169/239 (71%) of the diversified programs, at least one variant has an execution time distribution that is different compared to the execution time distribution of the original program. The result from this study encourages the composition of the variants to provide a resilient execution.

4.3 RQ3. To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?

Here we investigate the impact of the composition of program variants into multivariant binaries. To answer this research question, we create multivariant binaries from the program variants generated for Libsodium and QrCode corpora. Then, we deploy the multivariant binaries into the Edge and collect their execution times.

Execution times and timing side-channels.

We compare the execution time distributions for each program for the original and the multivariant binary. All distributions are measured on 100k executions of the program along all Edge platform nodes. We have observed that the distributions for multivariant binaries have a higher standard deviation of execution time. A statistical comparison between the execution time distributions confirms the significance of this difference (P-value = 0.05 with a Mann-Withney U test). This hints at the fact that the execution time for multivariant binaries is more unpredictable than the time to execute the original binary.

In Figure 4.3, each subplot represents the quantile-quantile plot [66] of the two distributions, original and multivariant binary. This kind of plots is used to compare the shapes of distributions, providing a graphical comparison of location, scale, and skewness for two distributions. The dashed line cutting the subplot represents the case in which the two distributions are equal, *i.e.,* for two equal distribution we would have all blue dots over the dashed line. These plots reveal that the execution times are different and are spread over a more extensive range of values than the original binary. The standard deviation of the execution time values evidences the latter, the original binaries have lower values while the multivariant binaries have higher values up to 100 times the original. Besides, this can be graphically appreciated in the plots when the blue dots cross the reference line from the bottom of the dashed line to the top. This is evidence that execution time is less predictable for multivariant binaries than original ones. This phenomenon is present because the choice of function variants is randomized at each function invocation, and the variants have different execution times due to the code transformations, i.e., some variants execute more instructions than others.

Answer to RQ3.

The execution time distributions are significantly different between the original and the multivariant binary. Furthermore, no specific variant can be inferred from execution times gathered from the multivariant binary. Consequently, attacks relying on measuring precise execution times [**?**] of a function are made a lot harder to conduct as the distribution for the multivariant binary is different and even more spread than the original one.

Conclusions

Our approach introduces static and dynamic, variants for up to 11.78% of the programs in our three corpora, increasing the original count of programs by 4.15 times. We highlighted the importance of complementing static and dynamic studies for programs diversification. Moreover, combining function variants in multivariant binaries makes virtually impossible to predict which variant is executed for a given query. We empirically demonstrate the feasibility and the application of automatically generating WebAssembly program variants.

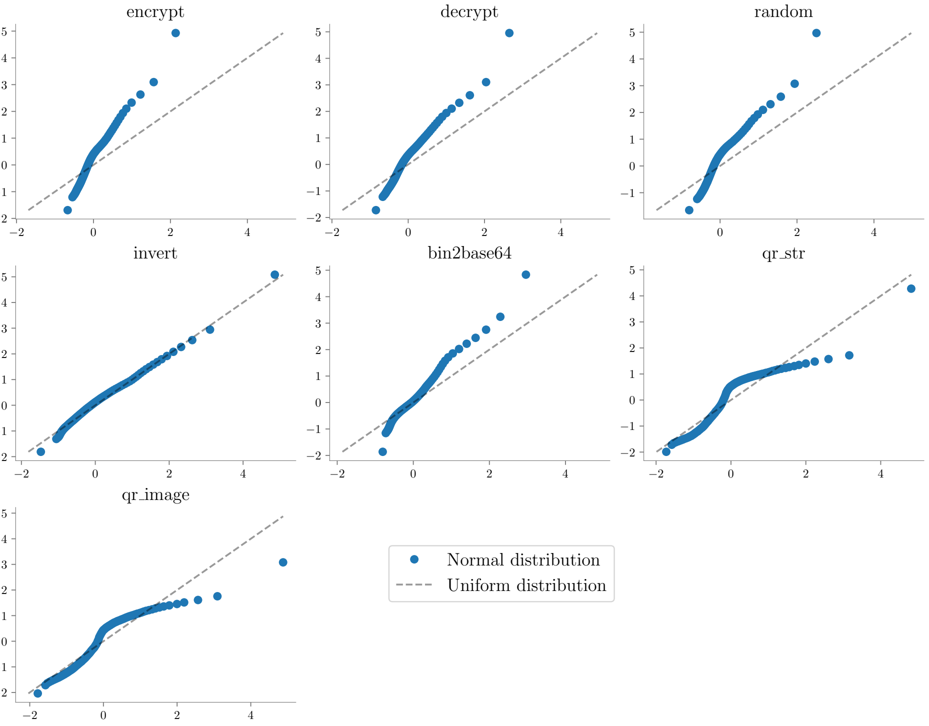


Figure 4.3: Execution time distributions. Each subplot represents the quantile-quantile plot of the two distributions, original and multivariant binary.

Chapter 5

Conclusions

Bibliography

[1] Romano,A., Lehmann,D., Pradel,M., and Wang,W. (2022). Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 1101–1116, Los Alamitos, CA, USA. IEEE Computer Society.

[2] Voulimeneas,A., Song,D., Larsen,P., Franz,M., and Volckaert,S. (2021). dmvx: Secure and efficient multi-variant execution in a distributed setting. In *Proceedings of the 14th European Workshop on Systems Security*, pages 41–47.

[3] Narayan,S., Disselkoen,C., Moghimi,D., Cauligi,S., Johnson,E., Gang,Z., Vahldiek-Oberwagner,A., Sahita,R., Shacham,H., Tullsen,D., et al. (2021). Swivel: Hardening webassembly against spectre. In *USENIX Security Symposium*.

[4] Lee,S., Kang,H., Jang,J., and Kang,B. B. (2021). Savior: Thwarting stack-based memory safety violations by randomizing stack layout. *IEEE Transactions on Dependable and Secure Computing*.

[5] Johnson,E., Thien,D., Alhessi,Y., Narayan,S., Brown,F., Lerner,S., McMullen,T., Savage,S., and Stefan,D. (2021). Sfi safety for native-compiled wasm. *NDSS. Internet Society*.

[6] Hilbig,A., Lehmann,D., and Pradel,M. (2021). An empirical study of real-world webassembly binaries: Security, languages, use cases. *Proceedings of the Web Conference 2021*.

[7] Cabrera Arteaga,J., Laperdrix,P., Monperrus,M., and Baudry,B. (2021b). Multi-Variant Execution at the Edge. *arXiv e-prints*, page arXiv:2108.08125.

[8] Cabrera Arteaga,J., Laperdrix,P., Monperrus,M., and Baudry,B. (2021a). Multi-Variant Execution at the Edge. *arXiv e-prints*, page arXiv:2108.08125.

[9] Cabrera Arteaga,J., Floros,O., Vera Perez,O., Baudry,B., and Monperrus,M. (2021). Crow: code diversification for webassembly. In *MADWeb, NDSS 2021*.

[10] (2021). National Cyber Leap Year.

[11] Xu,Y., Solihin,Y., and Shen,X. (2020). Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. In *Proc. of ASPLOS*, pages 987–1000.

[12] Shillaker,S. and Pietzuch,P. (2020). Faasm: Lightweight isolation for efficient stateful serverless computing. In *USENIX Annual Technical Conference*, pages 419–433.

[13] Runeson,P., Engström,E., and Storey,M.-A. (2020). *The Design Science Paradigm as a Frame for Empirical Software Engineering*, pages 127–147. Springer International Publishing, Cham.

[14] Lehmann,D., Kinder,J., and Pradel,M. (2020). Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association.

[15] Gadepalli,P. K., McBride,S., Peach,G., Cherkasova,L., and Parmer,G. (2020). Sledge: A serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*, page 265–279.

[16] Chen,D. and W3C group (2020). WebAssembly documentation: Security. Accessed: 18 June 2020.

[17] Bryant,D. (2020). Webassembly outside the browser: A new foundation for pervasive computing. In *Proc. of ICWE 2020*, pages 9–12.

[18] Roy,A., Chhabra,A., Kamhoua,C. A., and Mohapatra,P. (2019). A moving target defense against adversarial machine learning. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, page 383–388.

[19] Österlund,S., Koning,K., Olivier,P., Barbalace,A., Bos,H., and Giuffrida,C. (2019). kmvx: Detecting kernel information leaks with multi-variant execution. In *ASPLOS*.

[20] Aga,M. T. and Austin,T. (2019). Smokestack: thwarting dop attacks with runtime stack layout randomization. In *Proc. of CGO*, pages 26–36.

[21] Silvanovich,N. (2018). The problems and promise of webassembly. Technical report.

[22] Lu,K., Xu,M., Song,C., Kim,T., and Lee,W. (2018). Stopping memory disclosures via diversification and replicated execution. *IEEE Transactions on Dependable and Secure Computing*.

[23] Li,J., Zhao,B., and Zhang,C. (2018). Fuzzing: a survey. *Cybersecurity*, 1(1):1–13.

[24] Genkin,D., Pachmanov,L., Tromer,E., and Yarom,Y. (2018). Drive-by key-extraction cache attacks from portable code. *IACR Cryptol. ePrint Arch.*, 2018:119.

[25] Belleville,N., Couroussé,D., Heydemann,K., and Charles,H.-P. (2018). Automated software protection for the masses against side-channel attacks. *ACM Trans. Archit. Code Optim.*, 15(4).

[26] WebAssembly Community Group (2017b). WebAssembly Specification.

[27] WebAssembly Community Group (2017a). WebAssembly Roadmap.

[28] Sengupta,S., Vadlamudi,S. G., Kambhampati,S., Doupé,A., Zhao,Z., Taguinod,M., and Ahn,G.-J. (2017). A game theoretic approach to strategy generation for moving target defense in web applications. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, page 178–186.

[29] Haas,A., Rossberg,A., Schuff,D. L., Schuff,D. L., Titzer,B. L., Holman,M., Gohman,D., Wagner,L., Zakai,A., and Bastien,J. F. (2017). Bringing the web up to speed with webassembly. *PLDI*.

[30] Van Es,N., Nicolay,J., Stievenart,Q., D'Hondt,T., and De Roover,C. (2016). A performant scheme interpreter in asm.js. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, page 1944–1951, New York, NY, USA. Association for Computing Machinery.

[31] Koning,K., Bos,H., and Giuffrida,C. (2016). Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 431–442. IEEE.

[32] Couroussé,D., Barry,T., Robisson,B., Jaillon,P., Potin,O., and Lanet,J.-L. (2016). Runtime code polymorphism as a protection against side channel attacks. In *IFIP International Conference on Information Security Theory and Practice*, pages 136–152. Springer.

[33] Volckaert,S., Coppens,B., and De Sutter,B. (2015). Cloning your gadgets: Complete rop attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing*, 13(4).

[34] Morgan,T. D. and Morgan,J. W. (2015). Web timing attacks made practical. *Black Hat*.

[35] Kim,D., Kwon,Y., Sumner,W. N., Zhang,X., and Xu,D. (2015). Dual execution for on the fly fine grained execution comparison. *SIGPLAN Not.*

[36] Davi,L., Liebchen,C., Sadeghi,A.-R., Snow,K. Z., and Monrose,F. (2015). Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*.

[37] Crane,S., Homescu,A., Brunthaler,S., Larsen,P., and Franz,M. (2015). Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, pages 8–11.

[38] Baudry,B. and Monperrus,M. (2015). The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Comput. Surv.*, 48(1).

[39] Alon Zakai (2015). asm.js Speedups Everywhere.

[40] Agosta,G., Barenghi,A., Pelosi,G., and Scandale,M. (2015). The MEET approach: Securing cryptographic embedded software against side channel attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1320–1333.

[41] Okhravi,H., Rabe,M., Mayberry,T., Leonard,W., Hobson,T., Bigelow,D., and Streilein,W. (2013). Survey of cyber moving targets. *Massachusetts Inst of Technology Lexington Lincoln Lab, No. MIT/LL-TR-1166.*

[42] Homescu,A., Neisius,S., Larsen,P., Brunthaler,S., and Franz,M. (2013). Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. IEEE.

[43] Coppens,B., De Sutter,B., and Maebe,J. (2013). Feedback-driven binary code diversification. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–26.

[44] Maurer,M. and Brumley,D. (2012). Tachyon: Tandem execution for efficient live patch testing. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 617–630.

[45] Jackson,T. (2012). *On the Design, Implications, and Effects of Implementing Software Diversity for Security.* PhD thesis, University of California, Irvine.

[46] Sidiroglou-Douskos,S., Misailovic,S., Hoffmann,H., and Rinard,M. (2011). Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 124–134, New York, NY, USA. Association for Computing Machinery.

[47] Salamat,B., Jackson,T., Wagner,G., Wimmer,C., and Franz,M. (2011). Runtime defense against code injection attacks using replicated execution. *IEEE Trans. Dependable Secur. Comput.*, 8(4):588–601.

[48] Amarilli,A., Müller,S., Naccache,D., Page,D., Rauzy,P., and Tunstall,M. (2011). Can code polymorphism limit information leakage? In *IFIP International Workshop on Information Security Theory and Practices*, pages 1–21. Springer.

[49] Chen,T. Y., Kuo,F.-C., Merkel,R. G., and Tse,T. H. (2010). Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, 83:60–66.

[50] Salamat,B., Jackson,T., Gal,A., and Franz,M. (2009). Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 33–46.

[51] Maia,M. D. A., Sobreira,V., Paixão,K. R., Amo,R. A. D., and Silva,I. R. (2008). Using a sequence alignment algorithm to identify specific and common code from execution traces. In *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis (PCODA*, pages 6–10.

[52] Jacob,M., Jakubowski,M. H., Naldurg,P., Saw,C. W. N., and Venkatesan,R. (2008). The superdiversifier: Peephole individualization for software protection. In *International Workshop on Security*, pages 100–120. Springer.

[53] Salamat,B., Gal,A., Jackson,T., Manivannan,K., Wagner,G., and Franz,M. (2007). Stopping buffer overflow attacks at run-time: Simultaneous multi-variant program execution on a multicore processor. Technical report, Technical Report 07-13, School of Information and Computer Sciences, UCIrvine.

[54] Bruschi,D., Cavallaro,L., and Lanzi,A. (2007). Diversified process replicæ for defeating memory error exploits. In *Proc. of the Int. Performance, Computing, and Communications Conference*.

[55] Cox,B., Evans,D., Filipi,A., Rowanhill,J., Hu,W., Davidson,J., Knight,J., Nguyen-Tuong,A., and Hiser,J. (2006). N-variant systems: a secretless framework for security through diversity. In *Proc. of USENIX Security Symposium*, USENIX-SS'06.

[56] Pohl,K., Böckle,G., and Van Der Linden,F. (2005). *Software product line engineering: foundations, principles, and techniques*, volume 1. Springer.

[57] Bhatkar,S., Sekar,R., and DuVarney,D. C. (2005). Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the USENIX Security Symposium*, pages 271–286.

[58] Kc,G. S., Keromytis,A. D., and Prevelakis,V. (2003). Countering code-injection attacks with instruction-set randomization. In *Proc. of CCS*, pages 272–280.

[59] Bhatkar,S., DuVarney,D. C., and Sekar,R. (2003). Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the USENIX Security Symposium.*

[60] Barrantes,E. G., Ackley,D. H., Forrest,S., Palmer,T. S., Stefanovic,D., and Zovi,D. D. (2003). Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. CCS*, pages 281–289.

[61] Chew,M. and Song,D. (2002). Mitigating buffer overflows by operating system randomization. Technical Report CS-02-197, Carnegie Mellon University.

[62] Forrest,S., Somayaji,A., and Ackley,D. (1997). Building diverse computer systems. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133)*, pages 67–72.

[63] Cohen,F. B. (1993). Operating system protection through program evolution. *Computers & Security*, 12(6):565–584.

[64] Avizienis and Kelly (1984). Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17(8):67–80.

[65] Needleman,S. B. and Wunsch,C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. 48(3):443–453.

[66] Gnanadesikan,R. and Wilk,M. B. (1968). Probability plotting methods for the analysis of data. *Biometrika*, 55(1):1–17.

[67] Mann,H. B. and Whitney,D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, 18(1):50–60.